TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

BAKALÁŘSKÁ PRÁCE

Liberec 2011 **Jan Žmolík**

TECHNICKÁ UNIVERZITA V LIBERCI

Fakulta mechatroniky, informatiky a mezioborových studií

Studijní program: B2646 – Informační technologie
Studijní obor: 1802R007 – Informační technologie

Návrhové vzory pro WWW aplikace

Design patterns for web applications

Bakalářská práce

Autor: Jan Žmolík
Vedoucí práce: Mgr. Jiří Vraný, Ph.D.

V Liberci 10. května 2011

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky a mezioborových studií
Akademický rok: **2010/2011**

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(PROJEKTU, UMĚLECKÉHO DÍLA, UMĚLECKÉHO VÝKONU)

Jméno a příjmení:	**Jan ŽMOLÍK**
Osobní číslo:	**M08000189**
Studijní program:	**B2646 Informační technologie**
Studijní obor:	**Informační technologie**
Název tématu:	**Návrhové vzory pro WWW aplikace**
Zadávající katedra:	**Ústav nových technologií a aplikované informatiky**

Zásady pro vypracování:

1. Seznamte se s problematikou objektově orientovaného programování, zejména s použitím návrhových vzorů v jazyce PHP.
2. V teoretické části práce uveďte nejvýznamnější návrhové vzory používané či použitelné při programování WWW aplikací.
3. Získané teoretické znalosti využijte k vytvoření komunitního www portálu.

Rozsah grafických prací: **dle potřeby**

Rozsah pracovní zprávy: **cca 50 stran**

Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam odborné literatury:

[1] Rudolf Pecinovský: Návrhové vzory - 33 vzorových postupů pro objektové programování, Computer Press 2007, ISBN: 978-80-251-1582-4

[2] HERRINGTON, Jack D. IBM - United States [online]. 18 Jul 2006 [cit. 2010-10-14]. Five common PHP design patterns. Dostupné z WWW: http://www.ibm.com/developerworks/library/os-php-designptrns/.

[3] LAM, Jason. Dev Articles : Programming Help, Web Design Help, CSS Help [online]. 2003-05-12 [cit. 2010-10-14]. Introduction to Design Patterns Using PHP. Dostupné z WWW: http://www.devarticles.com/c/a/PHP/Introduction-to-Design-Patterns-Using-PHP/.

Vedoucí bakalářské práce: **Mgr. Jiří Vraný, Ph.D.**
Ústav nových technologií a aplikované informatiky

Datum zadání bakalářské práce: **15. října 2010**

Termín odevzdání bakalářské práce: **20. května 2011**

V. Kopecký

prof. Ing. Václav Kopecký, CSc.
děkan

L.S.

prof. Dr. Ing. Jiří Maryška, CSc.
vedoucí ústavu

V Liberci dne 15. října 2010

Prohlášení

Byl(a) jsem seznámen(a) s tím, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci (TUL) nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu TUL.

Užiju-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti TUL; v tomto případě má TUL právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Bakalářskou práci jsem vypracoval(a) samostatně s použitím uvedené literatury a na základě konzultací s vedoucím bakalářské práce.

Datum

Podpis

Poděkování

Děkuji Mgr. Jiřímu Vranému, Ph.D. za vedení bakalářské práce a za podnětné konzultace. Dále děkuji všem, kteří mě při tvoření této práce podporovali.

Abstrakt

Tato práce se zabývá návrhovými vzory a jejich využitím v oblasti webových aplikací. Práce je rozdělena do dvou hlavních částí. První část obsahuje seznámení s vybranými návrhovými vzory a jejich principy. V druhé části je využití těchto vzorů demonstrováno na webové aplikaci. Závěr práce obsahuje shrnutí poznatků z praktické části a uvádí výhody a nevýhody užití návrhový vzorů při tvorbě webových aplikací.

Aplikace, na které je předvedeno použití návrhových vzorů, je obsažena na přiloženém CD.

Abstract

This work deals with design patterns and their use in Web applications. The work is divided into two main parts. The first part includes introduction with the chosen design patterns and principles. The second part demonstrates using of these models on a web application. The conclusion summarizes the findings from the practical part and gives the advantages and disadvantages of using design patterns for creating web applications.

Application, which shows using of design patterns, is included on the CD.

Klíčová slova

Návrhové vzory, webové aplikace, PHP, objektově orientované programování

Keywords

Design patterns, web applications, PHP, object-oriented programming

Obsah

Seznam obrázků

1 Úvod

V dnešní době vzniká velmi mnoho různých softwarových aplikací. Během návrhu softwaru jsou vývojáři nuceni řešit netriviální problémy, které se obvykle opakují. Návrhové vzory nabízí doporučené obecné řešení těchto často se vyskytujících problémů. Dle [1] na straně 34 se dají návrhové vzory přirovnat k matematickým vzorcům. Zásadním rozdílem však je, že místo čísel se do návrhových vzorů dosadí třídy, objekty a rozhraní.

Návrhové vzory mají za účel usnadnit návrh aplikace a zlepšit její přehlednost. To je vhodné především pro komunikaci ve skupinách, jelikož je daleko jednodušší popsat řešení jedním nebo dvěma slovy, než složitě vysvětlovat řešení vlastní. Vzory také počítají s budoucím rozšiřováním aplikace.

V současnosti, kdy je internet téměř v každé domácnosti, je značný zájem o webové aplikace. Při jejich vytváření je možné využít návrhových vzorů. Tato práce by měla ukázat využití vybraných návrhových vzorů v jazyce PHP.

První část práce osahuje úvod do problematiky návrhových vzorů a podrobnější seznámení s pěti vybranými návrhovými vzory, popis jejich principu a problémy, které je vhodné pomocí nich řešit. Další návrhové vzory vhodné pro webové aplikace jsou uvedeny v krátké rešerši na konci první části.

Druhá část práce obsahuje předvedení vlastní implementace návrhových vzorů a shrnuje výhody a nevýhody práce s nimi v jazyce PHP.

Práce z části navazuje na ročníkový projekt. Zadáním tohoto projektu bylo vytvořit komunitní web, který obsahoval databázi hudebních nahrávek. Registrovaní uživatelé měli možnost tyto nahrávky bodově hodnotit a přidávat k nim komentáře. Původní kód, ve kterém se prolínalo HTML a PHP byl značně nečitelný. V rámci praktické části bakalářské práce bylo tedy využito teoretických poznatků k přepsání aplikace za pomocí návrhových vzorů.

2 Cíle práce

Cílem práce je seznámení s problematikou návrhových vzorů v prostředí webových aplikací. Řešitel provede výběr nejpoužívanějších a nejvhodnějších návrhových vzorů pro webové aplikace. Tyto představí a popíše jejich principy chování a fungování. Popíše při tom, také situace kdy je vhodné využít návrhových vzorů a jaké jsou jejich přednosti. Tyto poznatky následně vhodně využije při tvorbě vlastní webové aplikace. V podobě krátké rešerše uvede další návrhové vzory použitelné pro webové aplikace.

Teoretické poznatky řešitel vhodně využije a implementuje vybrané návrhové vzory při vytváření webové aplikace. Na této aplikaci by měl demonstrovat výhody a nevýhody, které použití návrhových vzorů v jazyce PHP přináší.

3 Představení nástrojů

3.1 Návrhový vzor

Návrhový vzor je obecný předpis řešení netriviálního opakujícího se problému. Vzory mají usnadnit již návrh aplikace a následnou implementací zlepšit přehlednost kódu a v mnoha případech zajistit rozšiřitelnost aplikace. Návrhové vzory jsou úzce spjaty s problematikou objektově orientovaného programování, protože popisují řešení právě pomocí tříd a objektů.

OOP je metodika vývoje softwaru, která se snaží popsat reálné prvky pomocí objektů a jejich metod. V OOP se využívá principů, jako je zapouzdření, dědičnost nebo polymorfismus. Návrhové vzory velkou mírou využívají těchto vlastností OOP.

Vzory jsou obvykle popisovány jazykem UML. UML je jazyk pro popis architektury, která využívá OOP. K vytvoření UML diagramů v této práci byly využity předdefinované šablony v softwaru Enterprise Architect 8.0 ve verzi trial.

3.2 PHP

PHP je skriptovací programovací jazyk. V dnešní době patří k nejpopulárnějším jazykům v oblasti webových aplikací. Má značnou podporu ze strany hostingů a pokročilejší programátoři mohou sáhnout po některém z mnoha PHP frameworků, které si kladou za cíl usnadnit vývoj. Pro podrobnější informace je možné navštívit oficiální stránku, viz. [2].

Pro napsání vlastní aplikace a demonstraci vzorů byl využit framework Nette ve verzi 2.0-aplha. Jedná se o český framework pro tvorbu webových aplikací v PHP 5. Mezi hlavní předností patří dle [3] zaměření na bezpečnost aplikace, využívání technologií AJAX, SEO nebo MVC nebo vnitřní ladící nástroje.

Framework byl zvolen z důvodu existence mnoha jednoduchých návodů, které umožnily rychlé nastudování základů frameworku. Dále je také vhodné zmínit velkou a aktivní komunitu v ČR. Více informací lze získat na oficiální stránce, viz. [4].

4 Návrhové vzory

4.1 Prvky vzorů

V této práci budou návrhové vzory specifikovány názvem, řešeným problémem, podmínkami, za kterých je možné vzor použít, řešením a výsledkem. Konkrétní příklady řešení budou uvedeny v praktické části práce.

4.2 Základní rozdělení vzorů

Vzory jsou dle [5] obvykle rozdělovány do tří základních skupin:

- Creational Patterns – vzory ovlivňující vytváření objektů. Tato skupina vzorů řeší výběr vhodných tříd pro vytvoření objektů. Také jsou zodpovědné za správný počet vytvořených objektů.
- Structural Patterns – vzory ovlivňující uspořádání tříd a komponent. Tato skupina vzorů má za účel zpřehlednění aplikace.
- Behavioral Patterns – vzory ovlivňující chování systému. Tato skupina vzorů řeší výběr správné činnosti systému na základě stavu systému.

4.3 Vybrané vzory

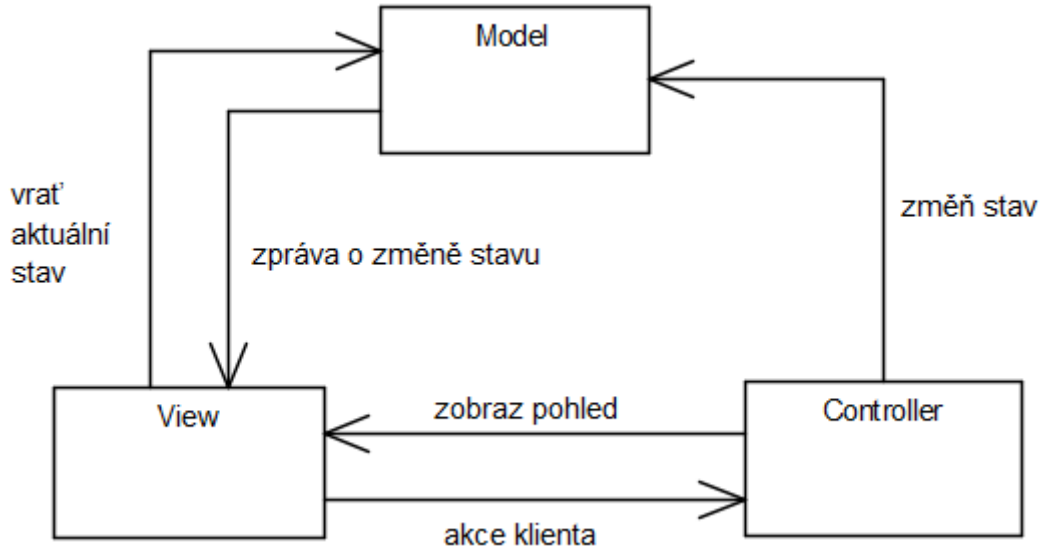
Dle názvů a obsahu článků [10] a [12], lze následující návrhové vzory označit, jako jedny z nejběžněji používaných vzorů při psaní webových aplikací.

4.3.1 MVC Pattern (Model-View-Controller)

MVC architektura má za úkol oddělit od sebe aplikační logiku, grafické prostředí aplikace a data. Rozděluje tedy vlastní aplikaci na tři samostatné moduly: datový model, řídící logiku a uživatelské rozhraní. Toto rozdělení je výhodné, protože velkou mírou přispívá k zpřehlednění kódu.

Pokud je projekt rozsáhlejší a pracuje na něm více programátorů zároveň, je výhoda rozdělení ještě silnější.

Řešení



Obr. 1 – MVC Pattern

Model: má přímý přístup k datům (obvykle k databázi) a umožňuje manipulaci s těmito daty.

View: zajišťuje prezentaci dat na straně uživatele (obsahuje grafické uživatelské prostředí)

Controller: komunikuje s Modelem a View. Předává mezi nimi data a zajišťuje správnou reakci na požadavky View (obvykle ze strany uživatele) a také správné chování View a Modelu.

Dle [6] se dá popsat cyklus probíhající v MVC architektuře následujícími kroky:

- Klient provede ve view akci
- Controller přebere informace a eventuelně data od View a předá je správnému Modelu
- Model na základě těchto informací a dat provede potřebné operace a změní svůj stav
- Controller informuje View o změně stavu Modelu a zadá View pokyn k vykreslení aktualizovaného Modelu
- View obdrží aktuální data a zobrazí je
- Po vykreslení View čeká na další akci ze strany klienta, po této akci se cyklus opakuje

Takto navržená architektura zajišťuje jednoduchou možnost úprav nebo rozšíření aplikace v budoucnu. Také umožňuje samostatný vývoj všech tří vrstev. Návrh aplikace bude flexibilnější a robustnější a bude snazší ho udržovat.

4.3.2 Strategy Pattern

Řešený problém

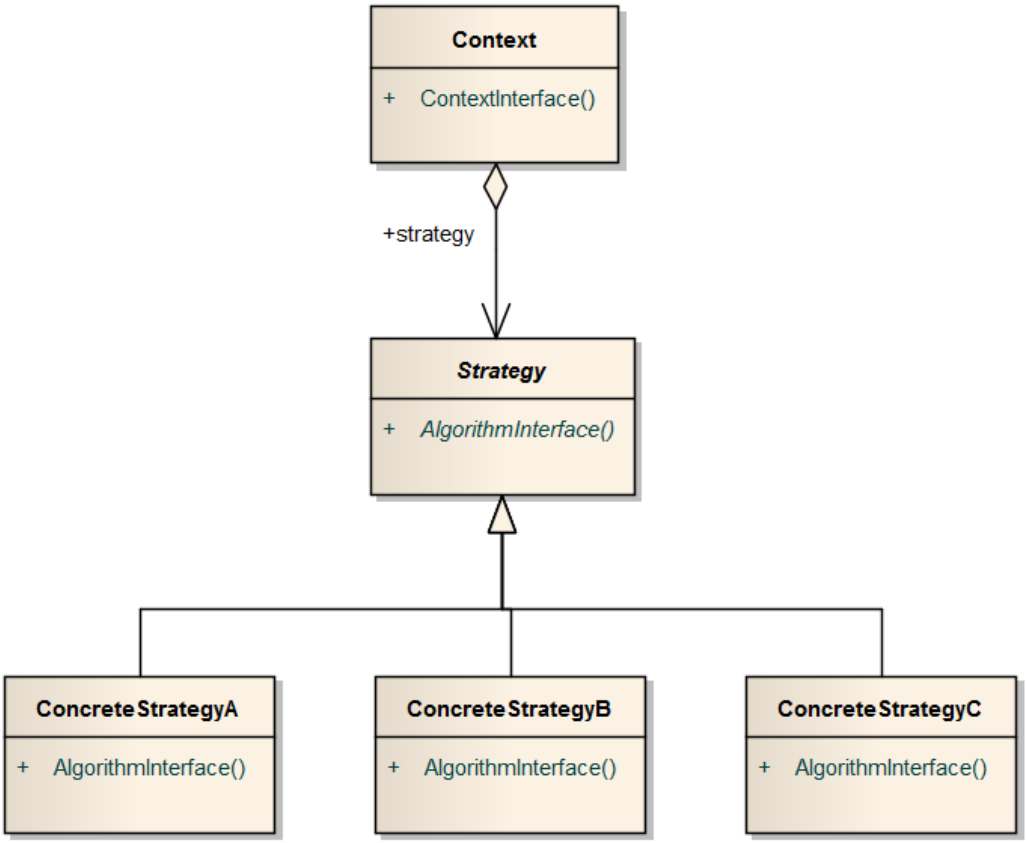
V [7] je uvedeno, že tento návrhový vzor umožňuje zapouzdřit algoritmy do samostatných objektů a jejich následnou záměnu. To, který algoritmus bude vybrán, je ovlivněno volbou klienta. Samotný výběr je ovšem řešen zcela nezávisle na klientovi.

Podmínky

Hlavní podmínkou pro použití tohoto vzoru je existence více podobných řešení jednoho problému. Jedná se o rodinu algoritmů, které rozdílným způsobem řeší stejnou funkci programu. Vzor má také zajistit znovu použitelnost algoritmů. Vzhledem k tomu, že algoritmy jsou implementovány jako jednotlivé objekty, je možné obejít se bez implementace jedné velké části kódu, která by obsahovala mnoho podmínek. Tím se zvyšuje čitelnost a právě znovu použitelnost jednotlivých algoritmů.

V praxi se tento vzor dá využít například implementací různých řadících algoritmů s tím, že si klient vybere, kterým algoritmem budou data seřazena. Dalším rozšířením může být možnost seřazení těchto dat sestupně nebo vzestupně.

Řešení



Obr. 2 – Strategy Pattern

Klient komunikuje s objektem Context. Ten obsahuje odkaz na objekt typu Strategy definující rozhraní, které implementují ConcreteStrategy. Díky polymorfismu se pak v odkazu, který obsahuje Context nachází některá z ConcreteStrategy. Ta se postará o správné zpracování požadavku od klienta. Obvykle je výběr ConcreteStrategy proveden na základě požadavku od klienta. V tomto případě většinou klient předává Contextu samotnou ConcreteStrategy, její název nebo nějaký speciální, programátorem definovaný, identifikátor. Je možné i druhé řešení výběru ConcreteStrategy, kdy Context sám vybere ConcreteStrategy, která je nejvhodnější pro zpracování požadavku.

Rozhraní třídy Strategy je navrženo tak, aby pomocí něj Context mohl používat ConcreteStrategy.

Dle [7] vzor funguje tak, že klient si vybere ConcreteStrategy, kterou předá Contextu. Context si uloží odkaz na ConcreteStrategy a umožní klientovi volání metod této strategie. V případě, že ke správnému vykonání algortimu jsou potřeba externí data od klienta, předá je ConcreteStrategy právě Context.

Výsledek

NV Strategy prakticky odděluje data (obsažená v Contextu) a logiku algoritmů (obsažené v jednotlivých ConcreteStrategy objektech). Data jsou následně použita pro zvolený algoritmus jako vstupní. Způsob, jakým je tento vzor navržen, umožňuje jednoduché rozšíření aplikace o nové algoritmy – přidáním nových ConctreteStrategy objektů. Toto rozšíření má vliv pouze na vlastní výběr algoritmu, kdy je nutné přidat možnost výběru nového objektu. Netýká se Contextu ani původních strategií.

4.3.3 State Pattern

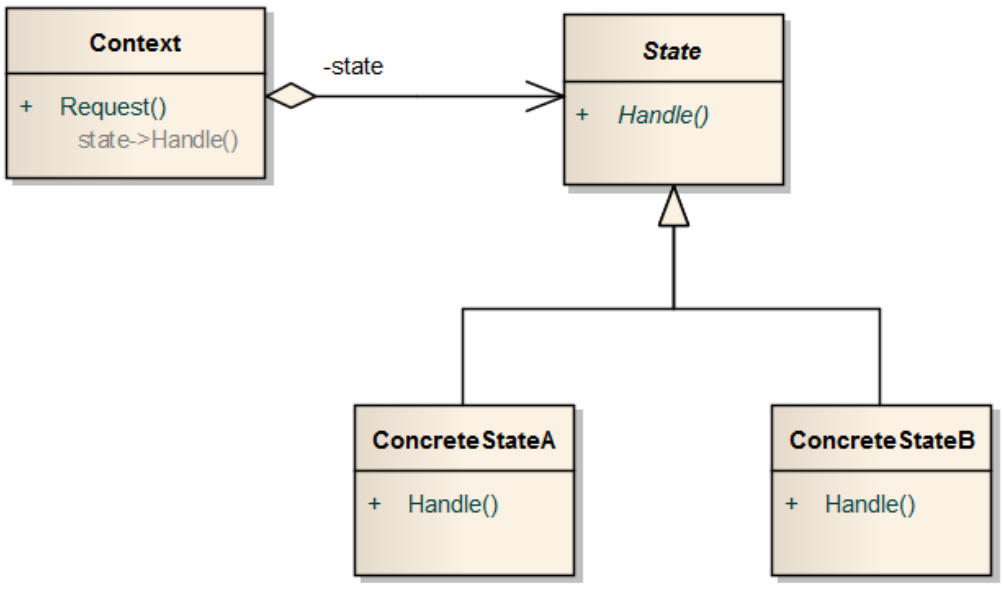
Řešený problém

[8] uvádí, že vzor je zaměřen na řešení problému, kdy je chování objektu závislé na jeho vnitřním stavu.

Podmínky

Je dán objekt, který má různé chování. Změny tohoto chování jsou závislé na vnitřním stavu daného objektu. V případě implementace sledování změn těchto stavů a následného výběru správného chování dochází k velkému nárůstu monolitické části kódu. Ten musí obsahovat mnoho podmínek a stává se tak nepřehledným. Vzor se snaží o zefektivnění tohoto řešení - vytváří třídy, které obsahují rozdílné chování objektu v závislosti na jeho aktuálním stavu.

Řešení



Obr. 3 – State Pattern

Vzor funguje na podobném principu ovládání jako Strategy Pattern. To znamená, že Context je objekt se kterým komunikuje klient. Obsahuje v sobě odkaz na objekt typu ConcreteState. Tím je určeno, který z ConcreteState objektů je aktivní a v jakém vnitřním stavu se objekt Context nachází. Třída State obsahuje rozhraní, které je implementováno ve třídách ConcreteState. Ve třídách ConcreteState je zapouzdřeno a definováno rozdílné chování objektu Context, v závislosti na jeho stavu. Context tak funguje jako prostředník mezi klientem a objektem ConcreteState.

Výsledek

Použitím NV State lze vytvořit systém, který zajistí rozdílné chování objektu v závislosti na stavu, v kterém se nachází. Tyto stavy jsou zapouzdřeny do samostatných tříd a díky tomu je kód významně zpřehledněn a přidání další Concrete State třídy je jednoduché. Dle [8] je další výhodou jasné určení třídy zodpovědné za chování. To vyplývá ze stavu objektu.

4.3.4 Factory Pattern

Řešený problém

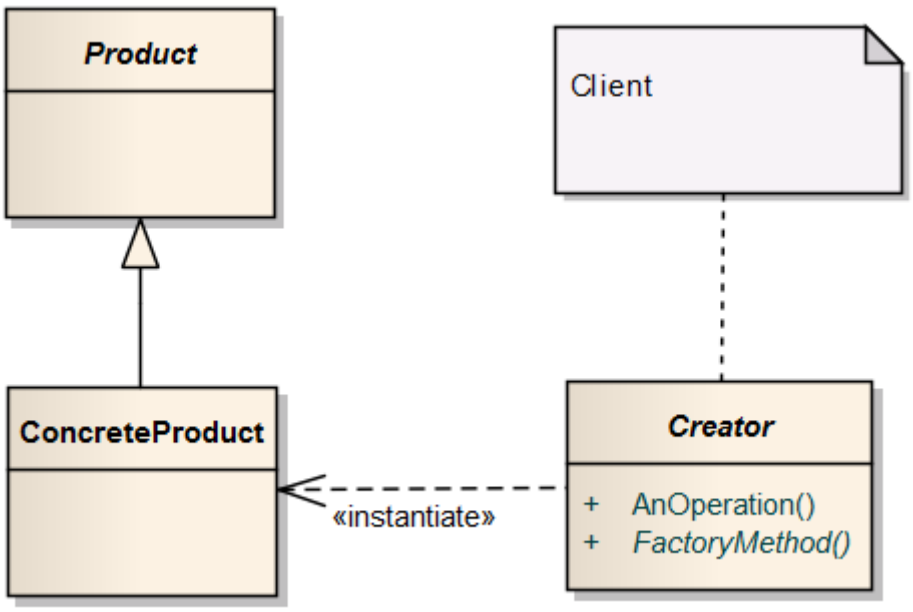
Vzor řeší problém, kdy je nutné rozhodnout o vytvoření instance konkrétní třídy až během vykonávání programu.

Podmínky

Dle [9] je základním předpokladem pro užití tohoto vzoru existence více různých tříd, které jsou většinou odvozeny od společného předka. Tyto třídy pak vykonávají nad daty rozdílné služby. Tovární metoda umožňuje za běhu programu zvolit jednu z těchto tříd, od které se vytvoří instance.

[9] dále uvádí, že je nutné rozhodnout, na jakém principu se bude vybírat třída, která je následně instanciována. Je tedy třeba definovat objekt, který je zodpovědný za způsob vytvoření instancí podřízených tříd.

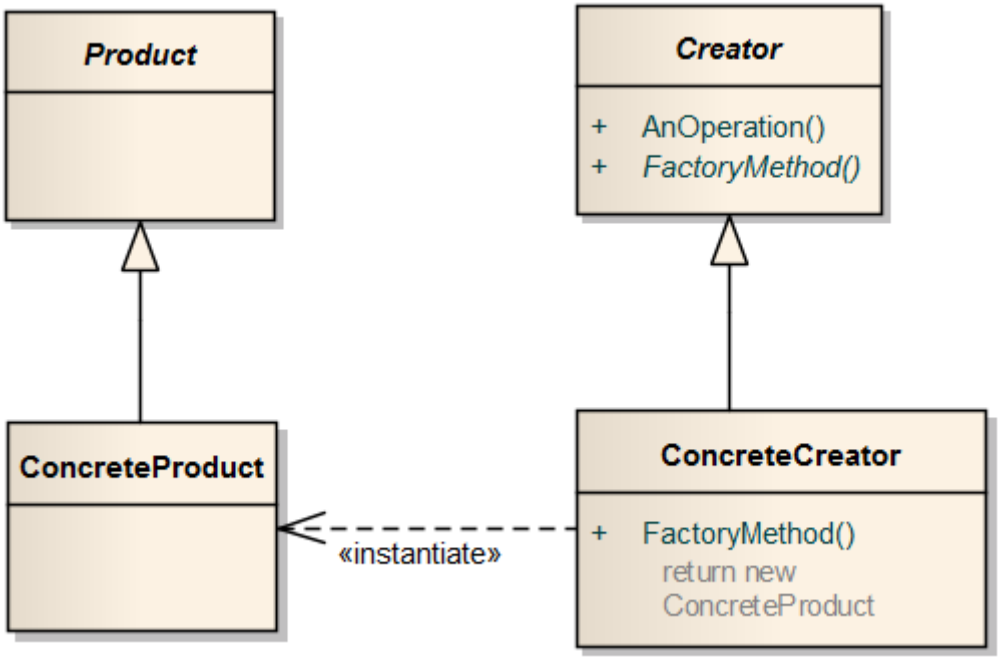
Řešení



Obr. 4 – Factory Pattern 1

NV Factory, lze řešit dvěma způsoby. První (jednodušší) způsob naznačený na Obr. 4 obsahuje třídy Creator, obecnou třídu Product a od ní odvozené třídy ConcreteProduct. Třída Product definuje rozhraní, které je implementováno odvozenými třídami.

Klient nekomunikuje přímo s produktem nebo konkrétními produkty, ale s třídou Creator. Ve chvíli, kdy klient zažádá třídu Creator o instanci Productu, Creator, na základě předaných parametrů a logiky, kterou má naimplementovanou, vybere nejvhodnější třídu Concrete Product a od té vrátí klientovi nově vytvořenou instanci. Klient tak vůbec neřeší výběr správné konkrétní třídy produktu a ve chvíli, kdy dostane odkaz na instanci ani neví jakého je typu a ani to pro něj není důležité z hlediska jeho dalšího fungování v systému.



Obr. 5 – Factory Pattern 2

Druhý způsob vyobrazený na Obr. 5 rozšiřuje ten první tak, že je použit jeden obecný Creator a od něj jsou v systému odvozeny třídy ConcreteCreator. Klient v tomto případě komunikuje s jednou ze tříd ConcreteCreator. Rozdíl oproti prvnímu řešení je, že v případě, kdy máme produkty rozděleny do skupin podle nějakého hlediska, mohou být ConcreteCreatory přidruženy právě k těmto skupinám. ConcreteCreator 1 pak vrací některý z ConcreteProductů ze skupiny 1. Je možné, aby každý z Creatorů vracel všechny ConcreteProducty, ale pak je výhodnější použít spíše první způsob.

Výsledek

Návrhový vzor zajišťuje odstínění klienta od logiky vytváření konkrétních objektů, se kterými pak klient dál pracuje. Tovární metoda, také může mít naimplementovanou logiku, která zajistí, předání již existujícího nevyužívaného objektu a nevytváří tak nový.

V [9] je dále uvedeno, že vzor Factory zajišťuje uzavření vytvářející logiky do jedné třídy a zvyšuje tím tak flexibilitu aplikace. Jakákoliv změna vytvářejících tříd se pak řeší na jednom místě, což je třída Product.

4.3.5 Singleton Pattern

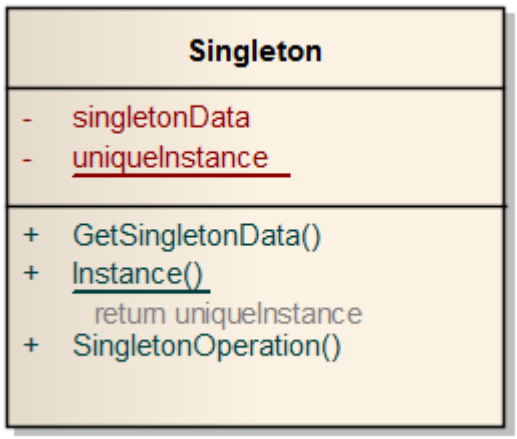
Řešený problém

Vzor zajišťuje vytvoření a následnou existenci pouze jediné instance určené třídy. Také zajišťuje globální přístup k této instanci.

Podmínky

Vlastní implementace návrhového vzoru Singleton je značně závislá na možnostech zvoleného programovacího jazyka. Je nezbytné při návhru a během implementace tohoto vzoru vzít v potaz všechny možnosti a omezení, které zvolený jazyk nabízí.

Řešení



Obr. 6 – Singleton Pattern

Jedním z možných řešení tohoto návrhového vzoru může být postup, kdy třída obsahuje privátní statickou proměnnou `uniqueInstance`, která bude stejného typu jako daná třída a bude obsahovat referenci na jedinou instanci. Dále je potřeba nastavit konstruktor třídy jako privátní a vytvořit statickou metodu vracející referenci obsaženou v proměnné `uniqueInstance`. Tato metoda zajistí vytvoření nové instance nebo v případě, že instance již byla vytvořena a uložena do proměnné `uniqueInstance`, tak vrátí referenci na tuto instanci. Toto řešení je výhodně i z toho důvodu, že k vytvoření instance dojde až ve chvíli, kdy je skutečně potřeba.

Dalším možným řešení je, že vytvoříme třídu, která bude final a její metody budou statické. Nastavením třídy jako final docílíme toho, že od třídy nebude možné dále dědit odvozené třídy. Vytvořením statických metod pak dosáhneme možnosti přistupovat k metodám přímo, aniž by bylo třeba vytvářet instanci. Nevýhodou tohoto způsobu ovšem je, že není vytvořena žádná instance a třída si tak nemůže zapamatovat svůj vnitřní stav a reagovat dle něj na chování programu. Výhodou tohoto řešení je, že se nemusíme starat o to, zda je vytvořena právě jedna instance – žádná se totiž nevytváří

a pracuje se přímo se statickou třídou a ta je, díky tomu že je final, v aplikaci právě přesně jednou.

Výsledek

Jelikož NV Singleton je možné implementovat různými způsoby, je nutné vybrat si takový, který vyhovuje konkrétně řešenému problému a potřebám vlastní aplikace. Také by neměl zvyšovat složitost samotné aplikace.

Nabízí jednoduché řešení v případech, kdy se připojujeme k databázi, přihlášení uživatele do systému, nebo v případě kdy je v objektu typu Singleton uloženo globální nastavení celé aplikace.

4.4 Další vzory

V této části jsou popsány obecné principy dalších pěti vzorů. Vzhledem k faktu, že při vývoji aplikace v praktické části nevnikly vhodné podmínky k užití těchto vzorů, jsou uvedeny pouze touto kratší formou.

Observer Pattern

[10] uvádí, že návrhový vzor Observer umožňuje předejít úzkému propojení mezi jednotlivými komponentami.

Vzor funguje tak, že definuje objekt, který je označen jako pozorovaný. Ten obsahuje metodu, pomocí které se může jiný objekt zaregistrovat jako pozorovatel. V případě, že se pozorovaný objekt změní, odešle všem svým zaregistrovaným pozorovatelům zprávu o této změně. To, jak naloží pozorovatelé s touto informací, je pro pozorovaný objekt nepodstatné. Výsledkem je způsob, kdy objekty mohou komunikovat mezi sebou a při tom nemusí nutně znát důvody vzniku této komunikace.

[11] také uvádí, že tento vzor je možné prakticky použít při implementaci MVC architektury.

Chain-Of-Command/Chain-Of-Responsibility Pattern

V [10] je napsáno, že tento vzor slouží ke zpracování zprávy, příkazu, žádosti nebo jiných požadavků pomocí sady objektů, které jsou v určené posloupnosti. Požadavek je předán prvnímu objektu v posloupnosti. Ten v případě, že může tento požadavek vykonat nebo obsloužit, ho vykoná a proces je zastaven. V opačném případě, je předán v posloupnosti následujícímu objektu. Ten obvykle má nadefinována vyšší

práva k manipulaci s požadavkem. A opět se opakuje situace, kdy je buď požadavek vykonán a proces ukončen, nebo je předán následujícímu objektu.

Vzor umožňuje přidání nebo odebrání objektů z posloupnosti, aniž by tím byly ovlivněny ostatní objekty.

Adapter Pattern

Návrhový vzor Adapter slouží podle [12] k převedení objektu jednoho typu na objekt jiného typu.

V [1] je na straně 261 uvedeno, že konverze proběhne způsobem, kdy je použito třídy Adapter ke změně rozhraní adaptovaného objektu na rozhraní, které očekává jiný objekt, případně klient. Klient pak může s objektem bez problémů komunikovat.

Vzor je vhodné použít například v situaci, kdy aplikace používá funkce externí knihovny. V případě, že je tato knihovna aktualizována dodavatelem na novou verzi a ta má jiné komunikační rozhraní, než verze používaná v naší aplikaci, vytvoříme pro část naší aplikace používající tuto knihovnu třídu Adapter.

Iterator Pattern

Jedná se o třídu, která zajistí průchod přes všechny prvky v sekvenčně uspořádaném seznamu. Třída Iterator přitom nezná implementaci jednotlivých prvků nebo samotného seznamu. Dle [1] na straně 204 obvykle Iterator nabízí dvě metody pro práci se seznamem. Jedna metoda vrací další prvek, druhá metoda zjišťuje, zda ještě nějaký další prvek existuje. Může také obsahovat další metody, například pro ovlivnění směru iterace, přesun ukazatele na jiný prvek, přidání nebo odebrání prvku.

Decorator Pattern

Návrhový vzor Decorator řeší dle [13], problém, kdy je třeba přidat nebo změnit instanci některé třídy vlastnosti a nevytvářet při tom novou třídu odvozením. Tímto způsobem se dá dynamicky skládat chování objektů.

[1] na straně 345 uvádí, že vzor funguje na principu „obalení“ jednoho objektu druhým. Konstruktor Decoratoru obdrží v parametru objekt, který je třeba rozšířit o nové funkce. Klient pak komunikuje s takto vytvořeným novým objektem. V případě, že volá metodu původního objektu, Decorator přenechá vykonání požadavku na původním objektu, naopak ve chvíli, kdy je třeba volat přidanou nebo změněnou funkci, vykonává jí právě Decorator.

5 Aplikace návrhových vzorů

Tato část práce obsahuje popis vlastní implementace vybraných návrhových vzorů, uvedení konkrétních problémů a zhodnocení výhod a nevýhod v těchto konkrétních případech.

Aplikace je komunitním web, který obsahuje informace o hudebních interpretech. Registrovaní uživatelé mají možnost hodnotit a komentovat nahrávky interpretů.

Zdrojové soubory jsou na přiloženém CD v adresáři BPAplikace\app. Zde se nachází adresářová struktura rozdělující jednotlivé části dle architektury MVP (viz. níže).

Jde o následující složky:

- models – obsahuje všechny modely (*.php soubory)
- presenters – obsahuje všechny presentery (*.php soubory)
- templates – obsahuje soubor @layout.latte, což je View pro základní vzhled aplikace. Dále obsahuje složky, které mají stejné pojmenování jako názvy Presenterů – každá složka obsahuje sadu příslušných View (*.latte soubry) pro daný Presenter.

Na webové adrese http://janzmolik-bp.g6.cz/BPrace/www/ je možné si aplikaci vyzkoušet. Pro přihlášení do administrátorské sekce lze použít přihlašovací jméno „Admin“ a jako heslo „heslo“.

5.1 MVC Pattern (Model-View-Controller)

Popis problému

Pokud vyvíjíme aplikaci, která je ovládána pomocí grafického rozhraní, vzniká potřeba jednoznačně oddělit logiku aplikace od grafického prostředí.

Začínající programátor v PHP, obvykle sáhne po některém ze starších, nicméně stále funkčních tutoriálů na výuku základů jazyka PHP. Nešvarem těchto výukových materiálů je, že jsou většinou psány stylem, který kombinuje HTML a PHP kód v jednom monolitickém celku. Potenciální student si pak obvykle tento návyk osvojí a kód, který vytvoří je pak hůře čitelný a špatně se v něm orientuje.

MVC architektura nabízí řešení, jak oddělit právě grafické prostředí a aplikační logiku. Navíc ještě také odděluje datovou část, tzn. reprezentaci dat v rámci aplikace.

Tímto oddělením značně zpřehlední kód – eliminace jednolitého celku kódu. Bude pak snazší kód číst a upravovat.

Použití vzoru

[14] uvádí, že framework Nette implementuje MVC v podobě MVP, kde P označuje Presenter. Nejzásadnějším rozdílem mezi MVC a MVP je podle [15] ten, že u MVC se o zachytávání uživatelského vstupu stará Controller, kdežto u MVP se o toto stará View, který obvykle po kliknutí myši volá některou z metod Presenteru.

V Presenteru je pro každý vykreslitelný pohled (View) definována metoda s názvem `renderXXX()`, kde XXX značí název aktuálního View. V případě, že uživatel chce zobrazit daný pohled je zavolána metoda z příslušného Presenteru, což zajistí framework.

Jako konkrétní příklad lze uvést sekci vyhledávání. V té je uživateli vykreslen formulář, pomocí něhož lze vyhledávat v databázi interprety a nahrávky. Poté co uživatel zadá vyhledávaný řetězec a stiskne tlačítko Odeslat je předán signál od View `HomepagePresenteru`, respektive jeho metodě. Tato metoda je definována u formuláře (viz. soubor HomepagePresenter.php, řádek 266 a Obr. 7).

```
$form->onSubmit[] = callback($this, 'zpracujVysledky');
```

Obr. 7 – Nastavení metody zpracující formulář

Na Obr. 8 lze vidět, že metoda `zpracujVysledky()` předá data modelu `VyhledavaniStrategyContextModel`, který se postará o komunikaci s databází a vyhledaná data předá zpět `HomepagePresenteru`.

```php
public function zpracujVysledky(AppForm $form)
{
  // vytvoreni session a klice k session
  // vytvoreni klice bude treba do ostre verze predelat dynamicky
  $session = Nette\Environment::getSession('StoredForm.Data');
  $key = "tohleJeMujKlicKSession";

  $data = $form->getValues();
  // vyhledani dat pomoci Strategy pattern
  $strategyContext = new VyhledavaniStrategyContextModel($data['hledanyTyp']
                                                        , $data['hledanyVyraz']);

  // ulozeni vyhledanych dat a typu do promenne kvuli jednodussimu prenosu
  $prepravka[0] = $strategyContext->getData();
  $prepravka[1] = $data['hledanyTyp'];
  $session[$key] = $prepravka;

  $this->redirect("vysledky");
}
```

Obr. 8 – Metoda zpracujVysledky()

Data jsou uložena do session a je proveden redirect na metodu `renderVysledky()` téhož Presenteru, viz. Obr. 8. Protože se jedná o renderovací metodu, je předán příkaz View vysledky, že má být vykreslen. Metoda zajistí předání dat z databáze. Předání probíhá pomocí přiřazení do proměnné – v Presenteru (soubor HomepagePresenter.php) na řádku 309 je vidět předání proměnné s daty. Ve View k nim přistupujeme pomocí proměnné `$data` (soubor vysledky.latte, řádek 6).

View se postará o to, aby data byla vykreslena správně, a nezajímá se, jaké byly výsledky hledání.

Zhodnocení

Využít MVC architekturu u webové aplikace je výhodné, pokud je vytvářen větší projekt a existuje předpoklad, že v budoucnu bude aplikace značně rozvíjena. Navíc pokud se programátor rozhodne využít některý z frameworků pro webové aplikace, obvykle se setká s tím, že je tato architektura již implementována v rámci frameworku.

Menší komplikace, v podobě prodloužení práce, může nastat v případě, kdy se programátor teprve s koncepcí MVC seznamuje. Je tedy nutné zohlednit vlastní schopnosti a při přechodu na MVC architekturu se nesnažit psát hned rozsáhlé projekty, kde může neznalost základní koncepce přinést zbytečné chyby, ale začít na jednodušších aplikacích.

V případě webových aplikací je nutné počítat s omezením plynoucím z komunikace pomocí HTTP (hypertext transfer protocol – komunikační protokol, pro

přenos webových stránek). Jelikož je tento protokol bezestavový, webová aplikace si nemůže předávat proměnné z jedné stránky na druhou. Je nutné zajistit toto předávání, aby bylo možné využít principů návrhových vzorů. V této práci byla zvolena metoda, přenášení proměnných a objektů pomocí session.

5.2 Strategy Pattern

Popis problému

Klient (uživatel) má možnost vyhledání záznamu v databázi. Má na výběr tři kritéria, dle kterých může vyhledávat: jméno nahrávky, jméno interpreta a země původu interpreta. Klient napíše do formuláře řetězec nebo část řetězce, který chce vyhledat a z roletkového menu zvolí kritérium, podle kterého chce výraz vyhledat. Následně je klientovi zobrazen výsledek hledání, což jsou nalezená data, nebo oznámení, že hledaný řetězec se v databázi nenachází.

Použití vzoru

Jelikož je zvolení kritéria vyhledávání ponecháno na klientovi a vyhledávání se pokaždé řeší trošku jiným způsobem, nabízí se právě pro tento účel návrhový vzor Strategy.

V případě, že se klient rozhodne vyhledávat, je mu zobrazen formulář, do kterého zadá vyhledávaný řetězec a zvolí kritérium, jaký typ dat chce vyhledat. Po odeslání jsou tyto informace předány metodě `HomepagePresenteru zpracujVysledky()`. Ta zajistí vytvoření instance modelu `VyhledavaniStrategyContextModel`. Jak je vidět v souboru HomepagePresenter.php na řádku 281 jsou konstruktoru předány informace, které zadal klient. V souboru VyhledavaniStrategyContextModel.php lze vidět od řádku 17, jak je proveden výběr a uložení správné strategie, na základě požadavku klienta. Odkaz na objekt obsahující zvolenou strategii si kontext uloží do privátní proměnné `$strategy`.

```
// metoda vracejici data z databaze na zaklade zvolene strategie
public function getData() {
  return $this->strategy->najdi();
}
```

Obr. 9 – Metoda getData()

V `HomepagePresenteru` je zavolána metoda kontextu `getData()`. V souboru VyhledavaniStrategyContextModel.php je na řádku 39 vidět (také viz. Obr. 9), že metoda `getData()`, vrací sadu dat, kterou získá od uložené strategie, kdy je

zavolána metoda `najdi()`. Ta provede příkaz na databázi, který vyhledá řetězec zadaný uživatelem. Na Obr. 10 lze vidět ukázku jedné třídy Concrete Strategy. Její metoda `najdi()` se postará o vyhledání záznamů v databázi a vrátí tento výsledek.

```
class CStrategyAlbum extends AbstractStrategy {

    public function __construct($hledam) {
        $this->hledam = $hledam;
    }

    public function najdi() {
        return dibi::fetchAll('
            SELECT [jmenoAlbum], [datumVydani], [idAlbum], [jmenoInterpret]
            FROM [alba]
            JOIN [interpreti]
            USING (idInterpret)
            WHERE jmenoAlbum LIKE %~like~
            ORDER BY jmenoAlbum', $this->hledam
        );
    }
}
```

Obr. 10 – Třída Concrete Strategy

Poté, co tímto způsobem obdrží `HomepagePresenter` výsledky, odešle View `vysledky` příkaz k vykreslení a data z databáze. Tento pohled zobrazí klientovi nalezená data. To zda bylo něco nalezeno či nikoliv již není starost vzoru Strategy, ale samotného View. Při přesměrování na pohled vykreslující nalezená data instance Kontextu zaniká a při novém vyhledávání je tedy celá procedura opakována od začátku.

Původní návrh vyhledávání počítal s implementací tří konkrétních strategií, aby každá mohla obsluhovat jedno kritérium. Během dalšího vývoje aplikace bylo třeba v administrátorské sekci vypisovat všechny nahrávky, jejichž názvy začínají stejným písmenem. Jelikož tato funkce odpovídá vyhledávání, nabízelo se jednoduché řešení vytvořit instanci kontextu vzoru Strategy a vyhledat pomocí něj potřebná data. ConcreteStretegy, která je zde kontextu předána byla implementována dodatečně a lze tím demonstrovat snadnou rozšiřitelnost aplikace o nové strategie. Aby kontext o této strategii „věděl“ bylo třeba pouze malé úpravy v jeho konstruktoru, přidáním case větve od řádku 30 v souboru VyhledavaniStrategyContextModel.php.

Zhodnocení

Takto implementovaný návrhový vzor Strategy nepočítá se znovupoužitím instance Kontextu, ta totiž zaniká při přesměrování na View s výsledky a při dalším vyhledávání se vytváří instance nová. Tímto způsobem nelze dynamicky přepínat mezi

strategiemi, což ovšem v tomto konkrétním případě nevadí. Ovšem ve chvíli, kdy by bylo třeba strategii změnit, bude nutné kontext zachovat a předat (např. session), protože jinak by docházelo ke stálému vytváření nových instancí.

V případě, kdy je nutné zachovat kontext, je nezbytné doimplementovat do kontextu jednoduchou metodu, která změní strategii na základě předaného parametru. Takovýto postup bude vhodný především v případě, že s výsledky hledání rovnou zobrazíme i formulář pro nové vyhledávání. Nebylo by již třeba vytvářet novou instanci kontextu, ale stačilo by pouze zavolat metodu `setNewStrategy()` a předat jí jako parametr novou strategii.

Je tedy třeba dát pozor především na to, jak a kde chceme tento vzor používat. Je nutné dopředu počítat s použitím prostředku pro předávání kontextu, i kdyby to nakonec nebylo nezbytné.

Je nutné uvést také poznámku o dynamickém a slabém typování jazyka PHP. Na základě těchto vlastností totiž není nezbytné definovat konkrétním strategiím společného předka a aplikace přesto bude fungovat. PHP umožňuje do proměnné `$strategy` třídy `VyhledavaniStrategyContextModel` vložit bez problémů prakticky cokoliv. To znamená, že nejen odkaz na některou z konkrétních strategií, ale třeba textový řetězec nebo číslo. Je nutné toto brát v potaz a v případě práce na projektu ve více lidech zajistit, aby nemohl být obsah proměnné změněn „někým z venku“.

5.3 State Pattern

Popis problému

Klient, v tomto případě uživatel s právy administrátora, má přístup do administrační části aplikace. V této sekci má možnost editovat informace v databázi. Může editovat nahrávky, interprety a některé vlastnosti u uživatelů (například přidělení vyšších práv, omezení přístupu, vygenerování nového hesla).

Pokaždé se jedná o úpravu dat v databázi, ale vždy o data jiného typu. Bude tedy vhodné využít jednoho objektu, který nabídne klientovi metody ke správě databázových dat. Tímto objektem je kontext z návrhového vzoru State, u kterého bude stav dynamicky měněn podle toho, která data zrovna administrátor/klient potřebuje upravit. Klient se tedy vůbec nezajímá o to, jak je zařízeno, že jsou vždy provedeny operace nad správnými daty. Toto je prováděno automaticky při přechodu do jiného stavu, právě změnou stavu v kontextu.

Použití vzoru

Pokud administrátor vstoupí do administrační sekce je mu jako úvodní nabídka zobrazena editace interpretů. V této části je v `AdminPresenteru` vytvořena instance objektu `AdministraceStateContextModel` (viz. soubor AdminPresenter.php, řádek 39). Instance kontextu je uložena do session aby ji bylo možné přenášet mezi jednotlivými částmi administrace. Třída `AdministraceStateContextModel` je implementována jako Singleton, což je rozvedeno dále v části Singleton PatternSingleton. Pomocí metody `setState()` je nastaven výchozí stav, který odpovídá editaci interpretů. To znamená, že metodě byla předána v parametru instance objektu `CStateInterpret`.

V případě, že administrátor chce editovat nahrávky nebo uživatele, klikne na příslušný odkaz v menu. Tím předá řízení jiné metodě z `AdminPresenteru`. V této metodě je kontextu nastaven požadovaný stav a tím se změní jeho funkce. Poté když admin bude upravovat některý záznam, budou volány metody odpovídajícího stavu.

Pokud tedy přejde na administraci nahrávek, bude u kontextu zavolána opět metoda `setState()`. V parametru bude předána instance třídy `CStateAlba` (viz. soubor AdminPresenter.php, řádek 73).

V každé sekci má administrátor možnost nechat si vypsat seznam záznamů, které mají stejné počáteční písmeno. V případě interpretů je tak zobrazena tabulka a u každého záznamu nabídnuta možnost smazání z databáze. Ve chvíli, kdy uživatel zadá volbu Smazat u některého ze záznamů je zavolána metoda `AdminPresenteru actionDelete()`, která zpracuje požadavek (soubor AdminPresenter.php, řádek 126). V metodě je otevřena session, která obsahuje kontext s aktuálním stavem a je zavolána metoda kontextu `smaz()`. Zde volání metod kontextu funguje na stejném principu jako u vzoru Strategy. To znamená, že v metodě kontextu je volána adekvátní metoda uloženého objektu stavu. Po vykonání akce je proveden redirect na původní výpis interpretů. Nyní se již smazaný záznam v databázi nenachází, a proto není ani ve výpise.

Zhodnocení

U tohoto vzoru v podstatě platí stejné výhody a nevýhody jako u předchozího vzoru Strategy Pattern. Rozdíl je v tom, že u tohoto vzoru je potřeba kontext předávat neustále, což u vzoru Strategy nebylo ve výše uvedeném případě nezbytné. Pokud by nebyl kontext vzoru State předáván, popřel by touto implementací princip a nebude

využit správným způsobem, kdy je stav měněn dynamicky za běhu programu u stejné instance. V takovém případě by byla instance vytvářena neustále a stav by byl nastavován pokaždé prakticky pouze jednou jakožto výchozí.

Stejně jako u vzoru Strategy je nutné díky slabému a dynamicky typovanému PHP zajistit, aby kontext vždy obsahoval referenci na správný typ objektu – tedy některý z konkrétních stavů.

5.4 Factory Pattern

Popis problému

U interpreta je na základě dat v databázi zobrazena jeho aktivita (rok založení, rok ukončení, a zda hraje, či nikoliv). Před samotným načtením dat z databáze a před vykonáním programového kódu, není možné určit, jakým způsobem má být naloženo s informacemi z databáze – jaká je má zpracovat třída. Je proto vhodné tyto informace předat tovární metodě třídy `AktivitaFactoryModel`, která na základě těchto dat vybere správný objekt a předá mu data k následnému zpracování.

Použití vzoru

V případě, že uživatel zadá příkaz k vykreslení stránky s interpretem je předáno řízení aplikace metodě `renderInterpret()` `HomepagePresenteru`. Zde jsou od modelu `InterpretModel` zprostředkovávajícího komunikaci s databází přebrány data o interpretovi na základě jeho id. Informace o aktivitě interpreta jsou následně předána jako parametr tovární metodě třídy `AktivitaFactoryModel`. Tovární třída vyhodnotí obdržená data a zvolí správný objekt, kterému předá informace ke zpracování. K dispozici jsou tři třídy odvozené od společného předka. Třídy `staleHraje` a `nehraje` zpracují data a vrátí správný údaj o aktivitě interpreta. V případě, že se v databázi nachází chybná data, zvolí tovární metoda třídu `chybaDat`. Ta předá textový řetězec s informací, že v databázi jsou chybová data a proto nemůže být aktivita zobrazena.

Zhodnocení

Funkce vzoru Factory je vhodné využít v případě, kdy před začátkem vykonávání programu nevíme, z jakých dat budeme objekt tvořit a co od něj následně budeme očekávat.

V tomto případě využití je objekt vytvořený tovární metodou využit jednou a následně zaniká. Není tedy možnost využít další funkci tohoto vzoru, která umožňuje vrátit vhodný objekt, který již v paměti existuje, ale není právě nijak využíván. Pokud by bylo třeba tuto funkci využít, bude nutné nějakým způsobem uchovávat instance tvořených objektů (například pomocí serializace s použitím databáze) a k těm umožnit tovární metodě přístup.

5.5 Singleton Pattern

Popis problému

Jak již bylo naznačeno v části State Pattern, je jeho kontext implementován zároveň jako Singleton. V administrační části je vhodné vytvořit pro administrátora (klienta) pouze jednu instanci kontextu, se kterou bude komunikovat po celou dobu editací a tím vytvořit jeden přístupový bod do administrace. Toho dosáhneme použitím vzoru Singleton.

Použití vzoru

Jelikož předávání kontextu ze vzoru State je zajištěno pomocí session, vytváří se instance pouze na jednom místě, a to v metodě `renderInterpreti()` `AdminPresenteru`. V souboru AdminPresenter.php na řádku 39 je vidět, že instance je vytvářena pomocí tovární metody `getInstance()` třídy `AdministraceStateContextModel` a ne pomocí konstruktoru.

Ta zjistí zda-li již instance kontextu nebyla vytvořena dříve a klient se pouze do této sekce nevrátil (soubor AdministraceStateContextModel.php, řádek 19). Pokud ne je v této metodě vytvořena instance Kontextu a ta je zároveň uložena do proměnné `$uniqueContext` (soubor AdministraceStateContextModel.php, řádek 13). Metoda vrátí instanci `AdminPresenteru`, kde s ní je dále pracováno.

Objekt implementovaný jako Singleton je nutné předávat pomocí session. V případě, že vytvoříme instanci v jedné sekci a objekt nepředáme, tak se bude opět vytvářet nová, jelikož ta původní po vykreslení View souvisejícím s aktuální metodou Presenteru zaniká. Jiná metoda, která používá stejný objekt, by tak nevěděla o tom, že již nějaká instance dříve existovala. Došlo by tím tak k neustálému vytváření nových instancí, což neodpovídá funkci návrhového vzoru Singleton.

Zhodnocení

U tohoto vzoru je třeba zajistit předávání vytvořené instance mezi jednotlivými částmi aplikace, například pomocí session. V případě, že nebudeme předávat instanci, ale pokaždé volat metodu `getInstance()`, nebude fungovat návrhový vzor správně. Místo toho, aby zajistil jednu jedinou instanci daného objektu v celé aplikaci, bude pokaždé vytvářet instanci novou. Pokud toto dodržíme, je možné následně používat Singleton k účelům, kdy je potřeba pouze jediné instance objektu. V tomto případě byl návrhový vzor Singleton užit v kombinaci s kontextem návrhového vzoru State Pattern. Tím bylo zajištěno, že v administraci nevznikne více než jedna instance kontextu.

Je nutné se opět zmínit o typování PHP. Jelikož je odkaz na instanci ukládán do proměnné `$uniqueInstance`, je nezbytné opět zajistit nemožnost modifikace této proměnné.

6 Závěr

Jak je patrné z dílčích zhodnocení v předchozí části, návrhové vzory jsou v prostředí webových aplikací užitečné. V případě, že si programátor zvolí jazyk PHP, je nutné dát pozor na jistá omezení a vlastnosti tohoto jazyka, stejně jako protokolu HTTP.

V první řadě se jedná o fakt, že HTTP je bezestavový a nedokáže tak přenášet mezi jednotlivými částmi aplikace proměnné, objekty a další stavové informace. Toto omezení je možné překonat pomocí nástrojů jazyka PHP, které umožňují práci s cookies nebo session. Další z možností, jak přenášet data je ukládání do databáze nebo do souboru. V této práci byla zvolena metoda předávání proměnných a objektů pomocí session, aby bylo možné aplikovat uvedené poznatky nezávisle na volbě případného frameworku, nebo i v případě, že žádný framework nepoužijeme.

Druhý problém může nastat v případě slabého a dynamického typování v jazyce PHP. Jak bylo uvedeno v částech Strategy Pattern a State Pattern, je nutné zajistit, aby proměnné, ve kterých se nachází odkazy na objekty, nemohly být volně měněny. V případě, kdy by toto nebylo zajištěno, může nastat situace, kdy bude obsah proměnné přepsán a nejen že proměnná již nebude obsahovat instanci některého z objektů, ale může dokonce obsahovat jednoduchý datový typ. Tím by byla značně narušena funkčnost aplikace a ve chvíli, kdy bude nutné zavolat metodu příslušného objektu, skončí vykonávání programu chybou.

Jestliže budeme brát zřetel na tyto dvě výrazné nevýhody, již od začátku návrhu aplikace, mohou být návrhové vzory využity stejně jako u klasických desktopových aplikací. Využití návrhových vzorů ve vhodných situacích s sebou přinese zpřehlednění a lepší čitelnost kódu a případné rozšiřování aplikace v budoucnu bude jednodušší.

Literatura

[1] PECINOVSKÝ, Rudolf. *Návrhové vzory*. Vydání první. Brno : Computer Press, 2007. 527 s. ISBN 978-80-251-1582-4.

[2] *PHP : Hypertext Preprocessor* [online].The PHP Group, 2001, Last updated: Sun May 15 13:02:21 2011 [cit. 2011-05-15]. Dostupné z WWW: <http://www.php.net>.

[3] GRUDL, David. *Nette Framework* [online]. 19. 1. 2010, 2. 2. 2011 [cit. 2011-05-16]. Hlavní přednosti. Dostupné z WWW: <http://nette.org/cs/hlavni-prednosti>.

[4] *Nette Framework* [online].Nette Foundation, 11. 12. 2008, 15. 5. 2011 [cit. 2011-05-16]. Dostupné z WWW: <http://nette.org/>.

[5] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 16.06.2005 [cit. 2011-05-11]. Základní návrhové vzory. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-prehled>.

[6] SEVRJUKOV, Alexandr. *PHP MVC framework* [online]. 2007 [cit. 2011-05-11]. Co je Model View Controller?. Dostupné z WWW: <http://pvc.boolean.cz/?section=basic&page=mvc>.

[7] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 07.02.2006 [cit. 2011-05-11]. Strategy Pattern. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-Strategy>.

[8] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 16.06.2005 [cit. 2011-05-11]. State Pattern. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-State>.

[9] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 16.06.2005 [cit. 2011-05-11]. Factory Pattern. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-Factory>.

[10] HERRINGTON, Jack D. *DeveloperWorks* [online]. 18 Jul 2006 [cit. 2011-05-11]. Five common PHP design patterns. Dostupné z WWW: <http://www.ibm.com/developerworks/library/os-php-designptrns>.

[11] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 16.06.2005 [cit. 2011-05-11]. Observer Pattern. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-Observer

[12] GOOD, Nathan A. *DeveloperWorks* [online]. 25 Mar 2008 [cit. 2011-05-11]. Five more PHP design patterns. Dostupné z WWW: <https://www.ibm.com/developerworks/opensource/library/os-php-designpatterns/>.

[13] DVOŘÁK, Miloš. *Objekty* [online]. 12.06.2005, Naposledy upraveno 16.06.2005 [cit. 2011-05-11]. Decorator Pattern. Dostupné z WWW: <http://objekty.vse.cz/Objekty/Vzory-Decorator>.

[14] GRUDL, David. *Nette Framework* [online]. 23. 6. 2008, 22. 1. 2010 [cit. 2011-05-11]. Model-View-Presenter (MVP). Dostupné z WWW: <http://doc.nette.org/cs/model-view-presenter>.

[15] BERNARD, Borek. *Zdroják* [online]. 11. 5. 2009 [cit. 2011-05-11]. Prezentační vzory z rodiny MVC. Dostupné z WWW: <http://zdrojak.root.cz/clanky/prezentacni-vzory-zrodiny-mvc/>.

Příloha A

Obsah přiloženého CD

Adresář BP2011 – Obsahuje vlastní text bakalářské práce ve formátu pdf

Adresář BPAplikace – Obsahuje zdrojové kódy webové aplikace